(Near) Real time analytics with Apache Spark and Apache Cassandra

Christopher Batey

@chbatey

Who am I?

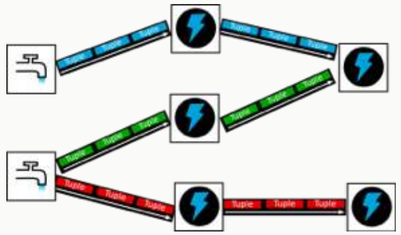
- Built a a lot of systems with Apache Cassandra at Sky
- Work on a testing library for Cassandra
- Help out Cassandra users
- Twitter: @chbatey

Agenda

- Motivation
- Cassandra
  - Replication
  - Fault tolerance
  - Data modelling
- Spark
  - Use cases
  - Stream processing
- Time series example: Weather station data

OLTP

OLAP



Batch

Weather data streaming

DATASTAX

Incoming weather events

Producer

Apache Kafka

Consumer

Spark Streaming

Dashboard

DATASTAX

Killr Weather Stations

ID	Name	Country Code	Call Sign	Latitude	Longitude
408930:99999	JASK	IR	OIZJ	25.65	57.767
725500:14942	OMAHA EPPLEY AIRFIELD	US	KOMA	41.317	-95.9
725474:99999	CRESTON	US	KCSQ	41.017	-94.367
480350:99999	LASHIO	BM	VBLS	22.933	97.75
719380:99999	COPPERMINE AIRPORT	CN	CYCO	67.817	-115.15
992790:99999	ENVIRONM BUOY 44008	US	DB279	40.5	-69.467
85120:99999	PONTA DELGADA/NORDE	PO	LPPD	37.733	-25.7
150140:99999	BAIA MARE	RO	LRBM	47.667	23.583
435330:99999	HANIMADU	MV		6.733	73.15
536150:99999	YINCHUAN (CITY)	CI		38.467	106.267

« 1 2 3 4 5 6 7 ... 1971 »

10 25 50 100

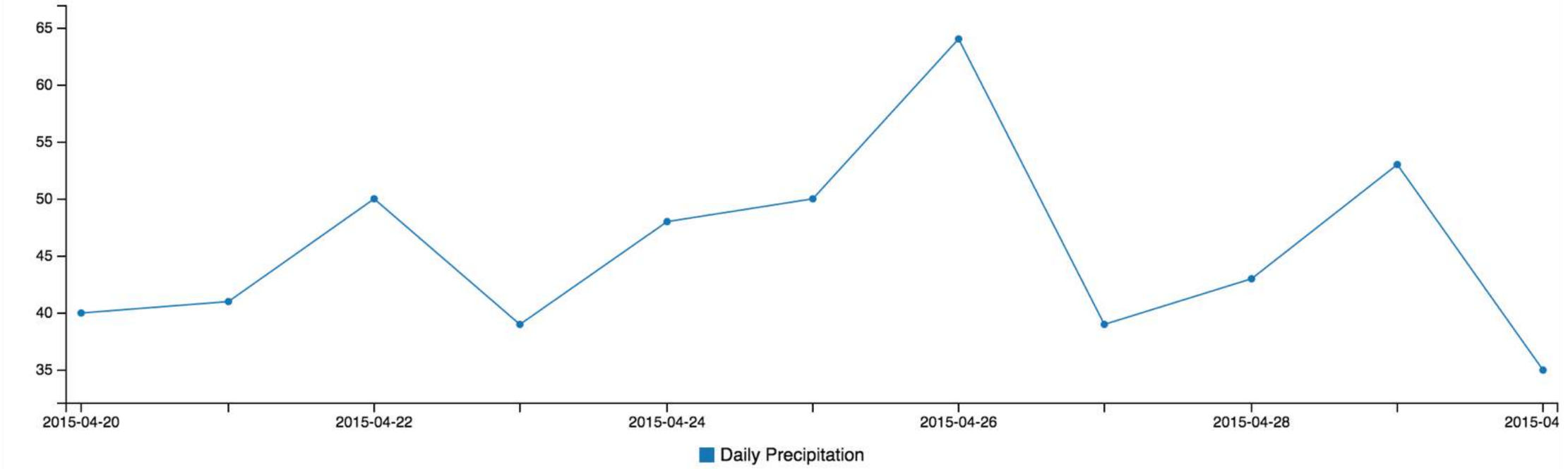
@chbatey

Killr Weather Stations

JASK (408930:99999)

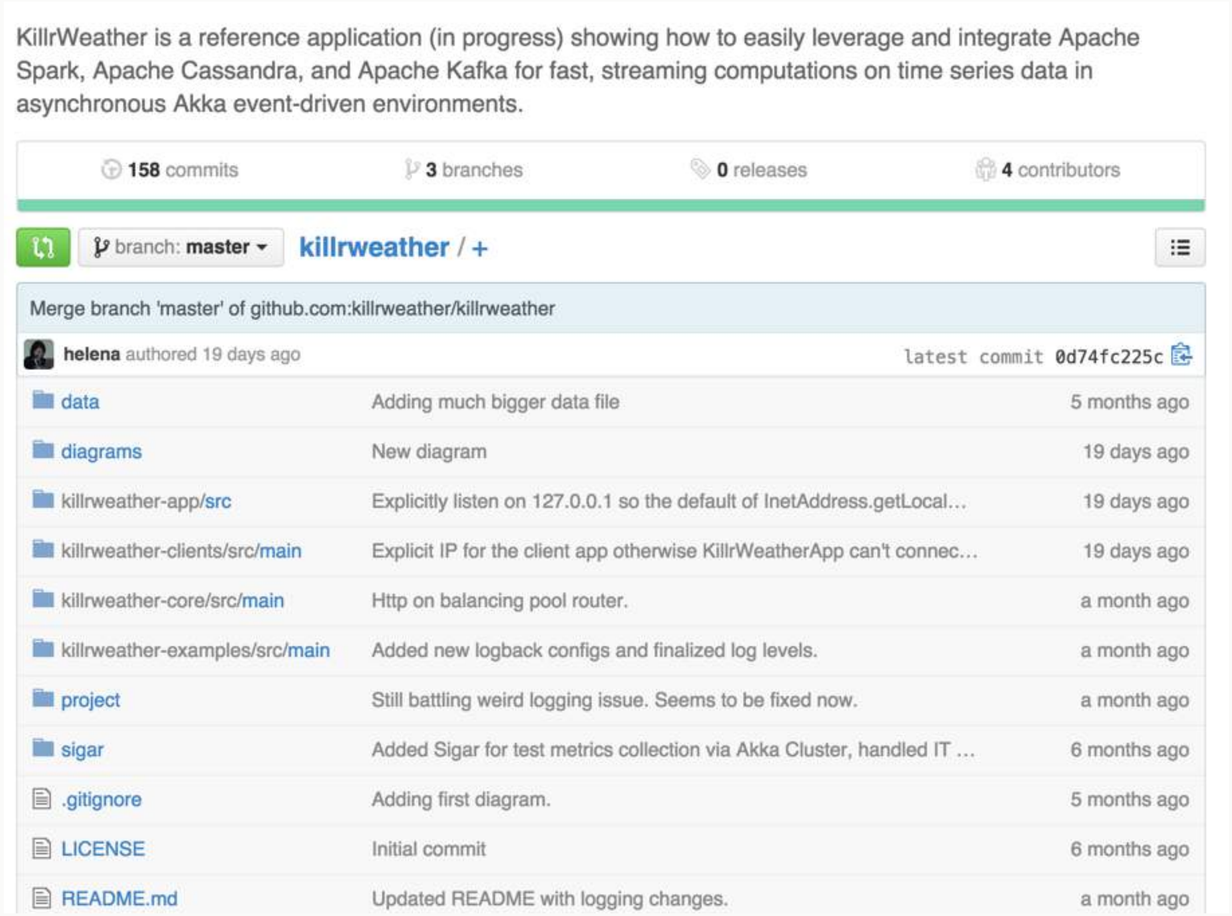
Weather Data Load Generation

Daily precipitation



Run this your self

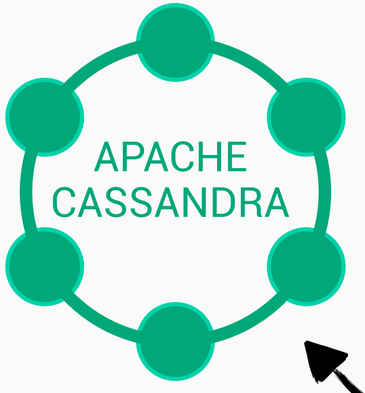
- https://github.com/killrweather/killrweather



DATASTAX

First we need a database

Cassandra for Applications

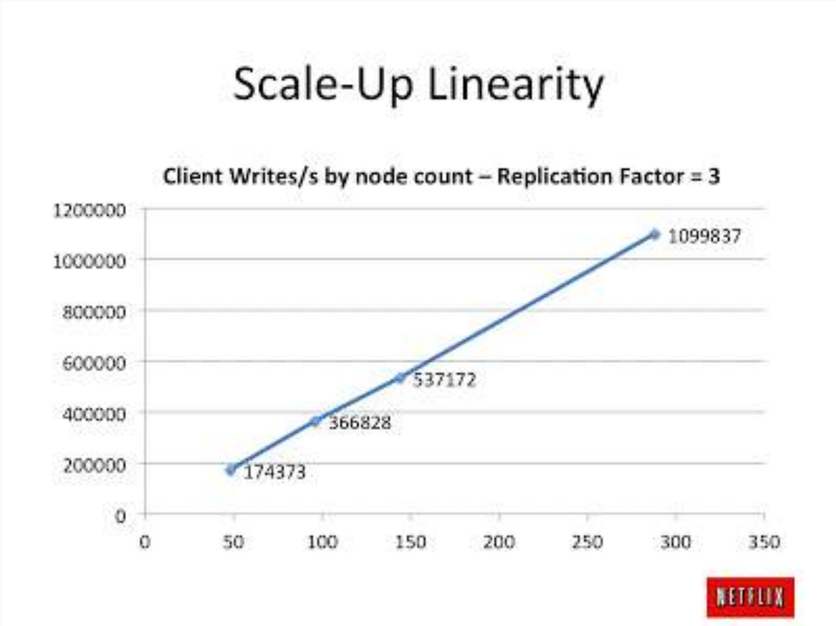


Common use cases

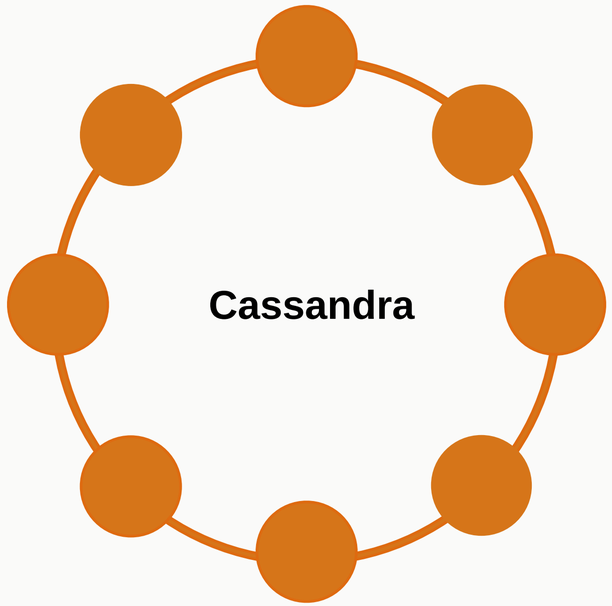
- Ordered data such as time series
  - Event stores
  - Financial transactions
  - IoT e.g Sensor data

Common use cases

- Ordered data such as time series
  - Event stores
  - Financial transactions
  - IoT e.g Sensor data
- Non functional requirements:
  - Linear scalability
  - High throughout durable writes
  - Multi datacenter including active-active
  - Analytics without ETL

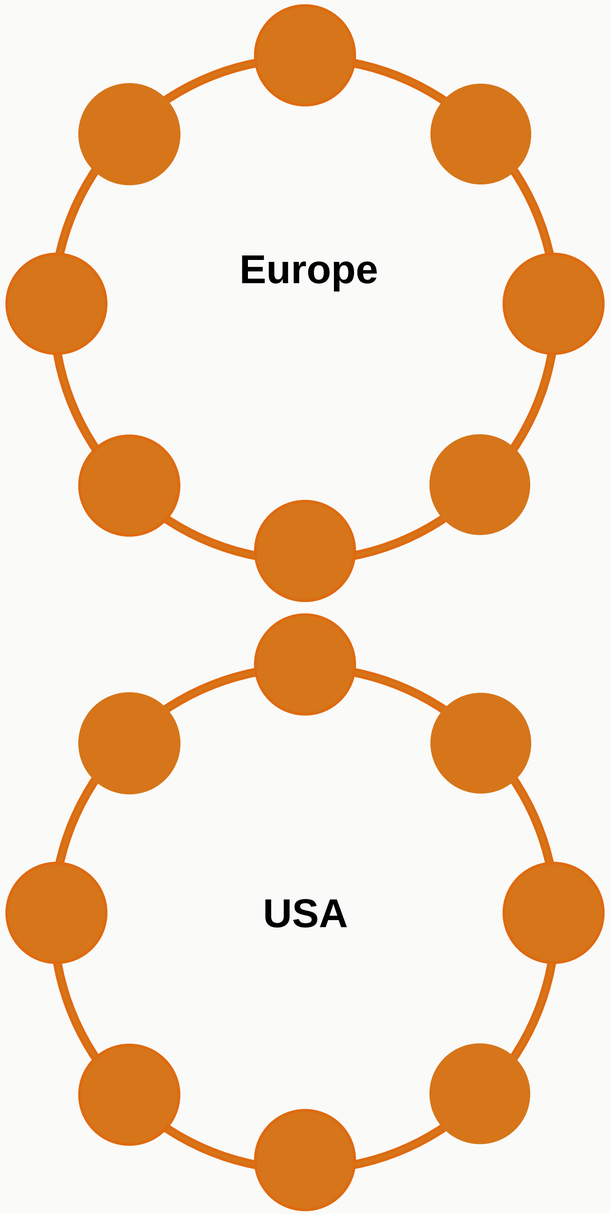


Cassandra



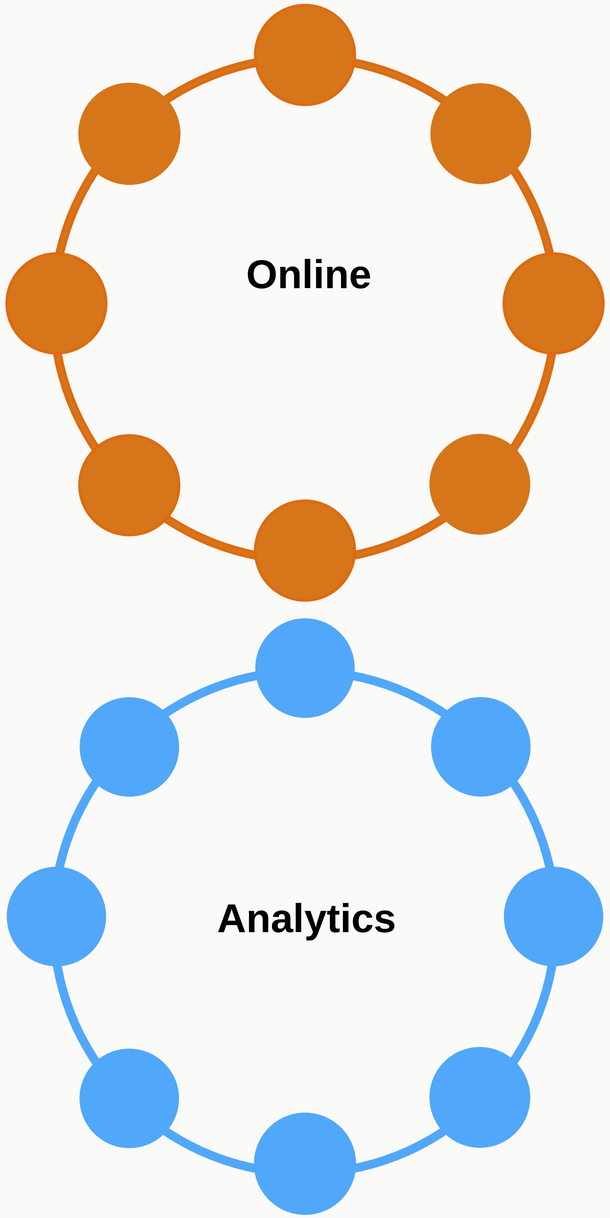
- Distributed masterless database (Dynamo)
- Column family data model (Google BigTable)

Datacenter and rack aware



- Distributed master less database (Dynamo)
- Column family data model (Google BigTable)
- Multi data centre replication built in from the start

Cassandra



- Distributed master less database (Dynamo)
- Column family data model (Google BigTable)
- Multi data centre replication built in from the start
- Analytics with Apache Spark

Dynamo 101

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

Dynamo 101

- The parts Cassandra took
  - Consistent hashing
  - Replication
  - ~~Gossip~~
  - ~~Hinted handoff~~
  - ~~Anti-entropy repair~~
- And the parts it left behind
  - Key/Value
  - Vector clocks

Picking the right nodes

- You don’t want a full table scan on a 1000 node cluster!
- Dynamo to the rescue: Consistent Hashing

Murmer3 Example

- **Data:**

jim	age: 36	car: ford	gender: M
carol	age: 37	car: bmw	gender: F
johnny	age: 12	gender: M	
suzy:	age: 10	gender: F	

- **Murmer3 Hash Values:**

Primary Key	Murmur3 hash value
jim	350
carol	998
johnny	50
suzy	600

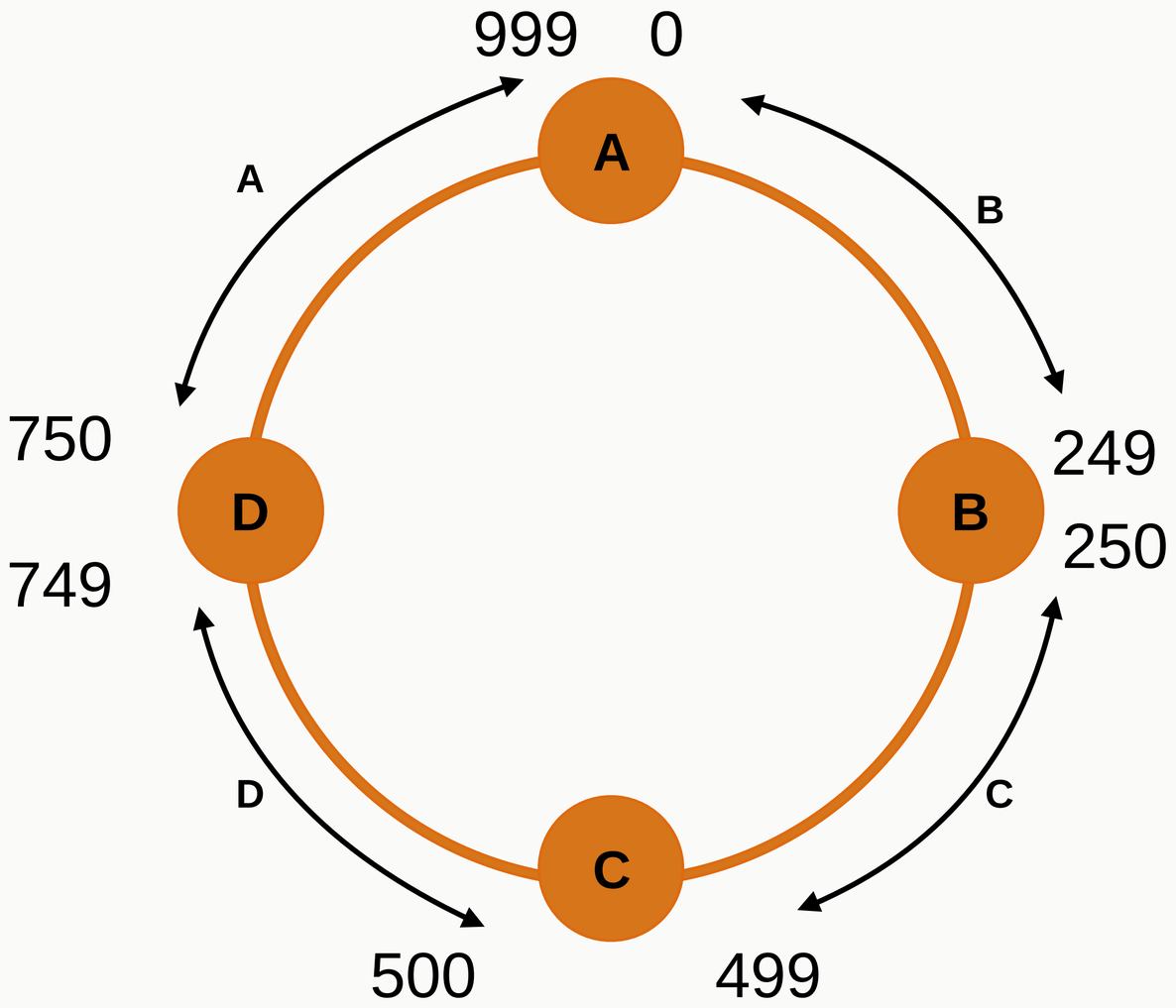
Real hash range: -9223372036854775808 to 9223372036854775807

Murmer3 Example

Four node cluster:

Node	Murmur3 start range	Murmur3 end range
A	0	249
B	250	499
C	500	749
D	750	999

Pictures are better



Murmer3 Example

Data is distributed as:

Node	Start range	End range	Primary key	Hash value
A	0	249	johnny	50
B	250	499	jim	350
C	500	749	suzy	600
D	750	999	carol	998

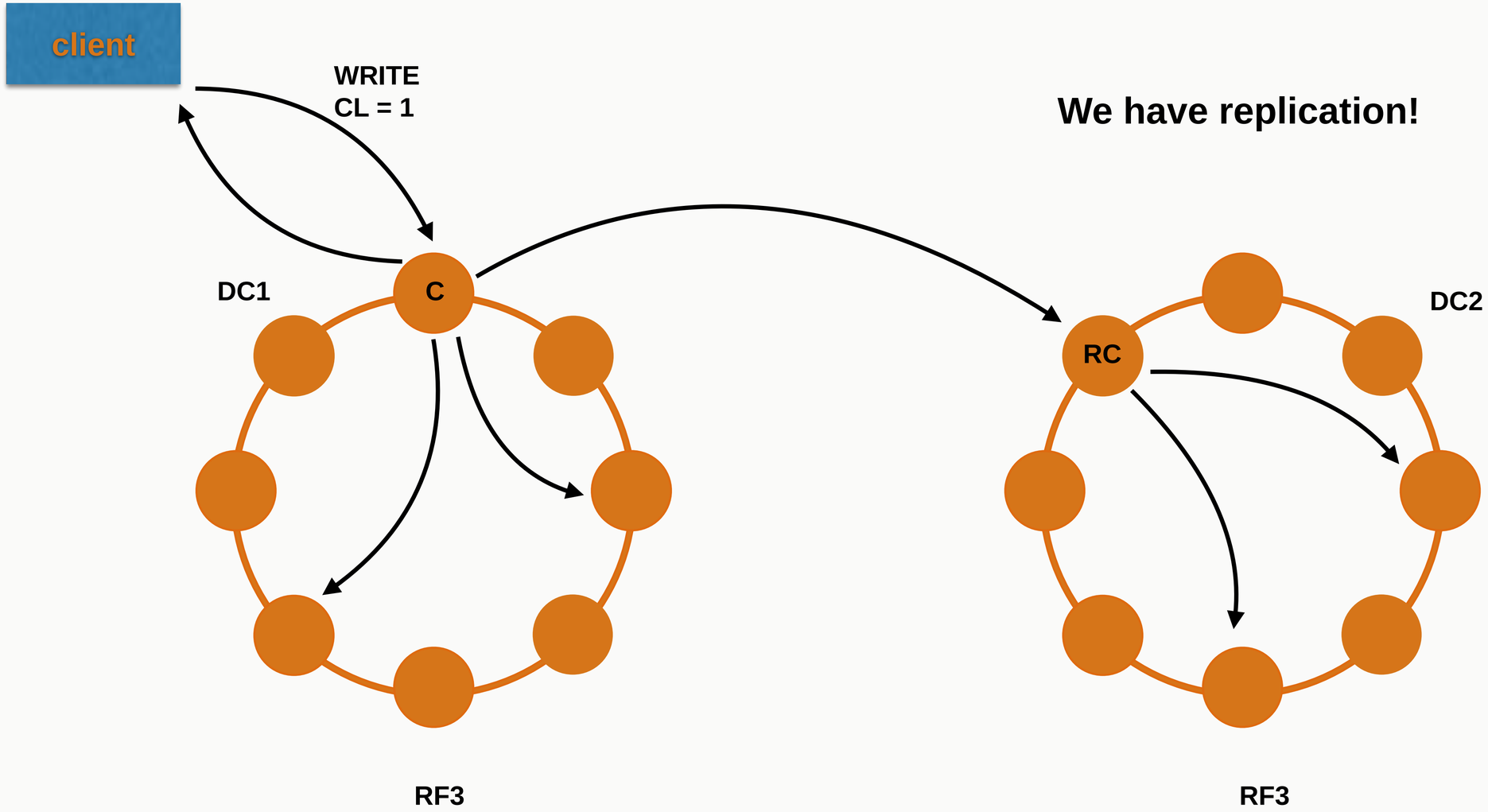
DATASTAX

Replication

Replication strategy

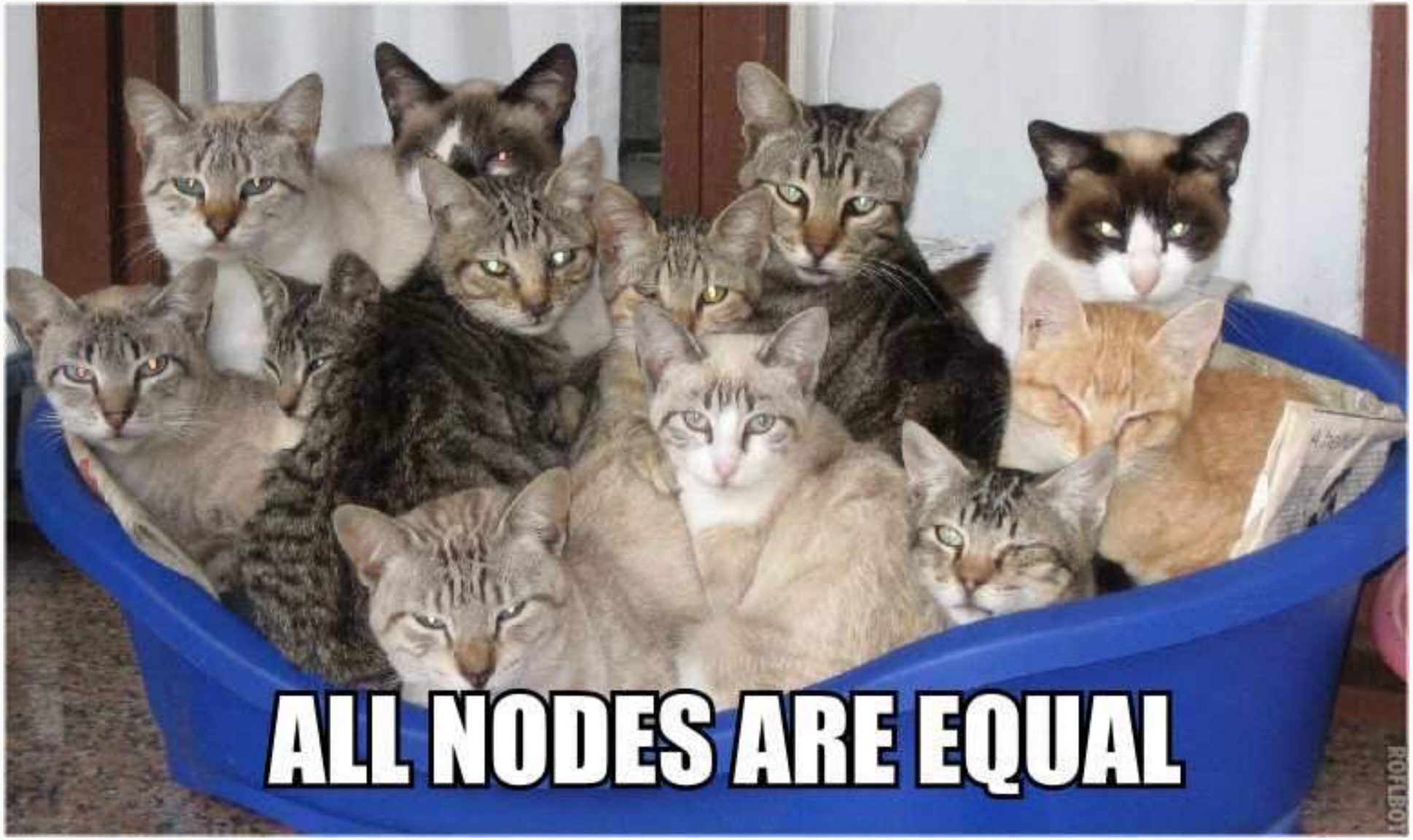
- NetworkTopology
  - Every Cassandra node knows its DC and Rack
  - Replicas won’t be put on the same rack unless Replication Factor > # of racks
  - Unfortunately Cassandra can’t create servers and racks on the fly to fix this :(

Replication



Tunable Consistency

- Data is replicated N times
- Every query that you execute you give a consistency
  - ALL
  - QUORUM
  - LOCAL_QUORUM
  - ONE
- **Christos Kalantzis** Eventual Consistency != Hopeful Consistency: http://youtu.be/A6qzx_HE3EU?list=PLqcm6qE9lgKJzVvwHprow9h7KMpb5hcUU


ALL NODES ARE EQUAL

DATASTAX

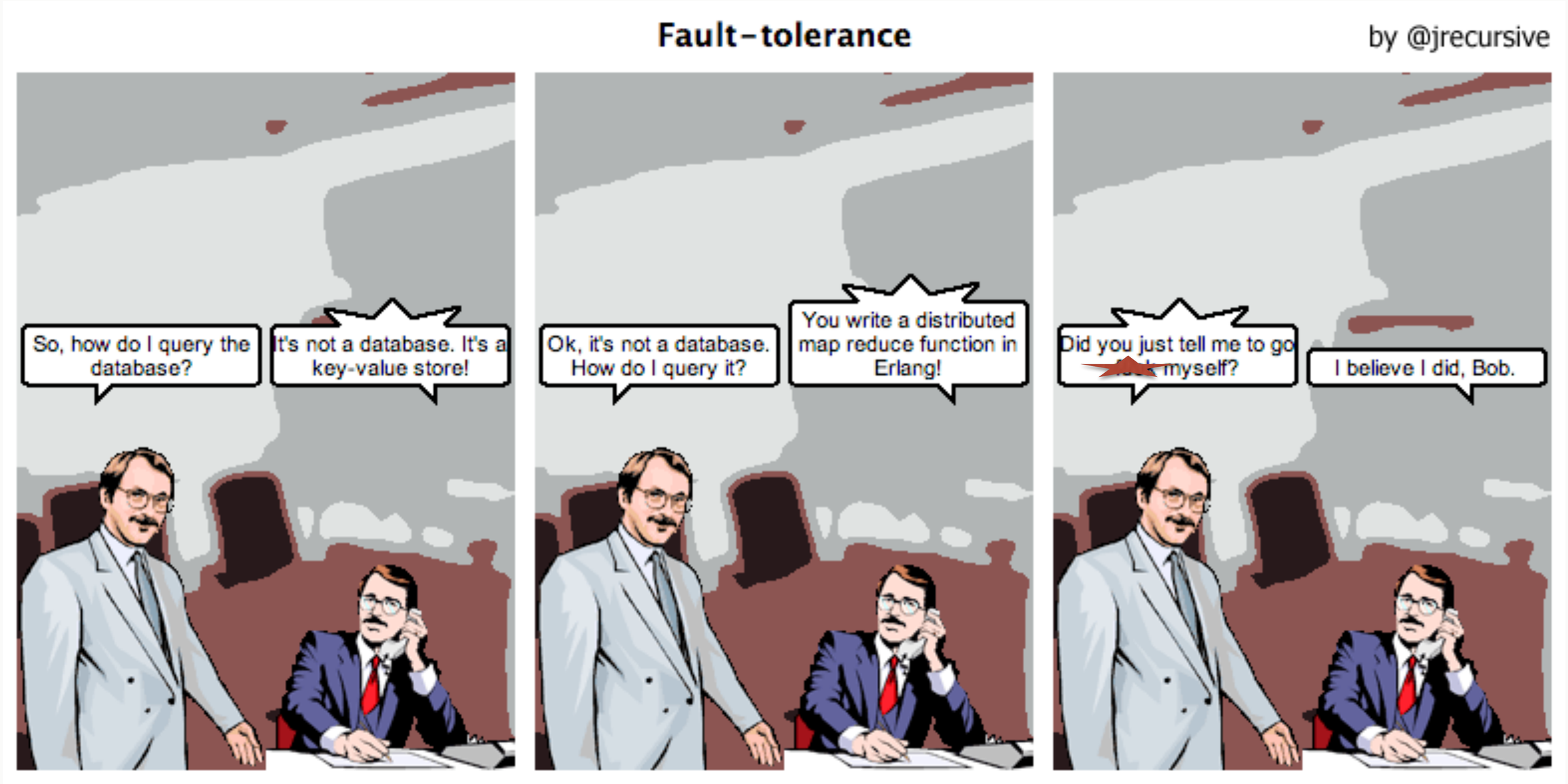
Next: meaning

---MORE---

wsid	year	month	day	hour	dewpoint	one_hour_precip	pressure	six_hour_precip	sky_condition	sky_condition_text	temperature	wind_direction	wind_speed
100460:99999	2015	5	2	0	-3.9	1.109	1020.4	5	2	1.108983997363286	27.422	270	4.6
100460:99999	2015	5	1	23	-3.9	0.046111	1020.4	5	2	0.046110866849417564	18.299	270	4.6
100460:99999	2015	5	1	22	-3.9	1.9641	1020.4	5	2	1.9641149108243532	18.386	270	4.6
100460:99999	2015	5	1	21	-3.9	2.4921	1020.4	5	2	2.492128263027498	19.268	270	4.6
100460:99999	2015	5	1	20	-3.9	0.24211	1020.4	5	2	0.24210684947010008	22.266	270	4.6
100460:99999	2015	5	1	19	-3.9	2.9651	1020.4	5	2	2.965103609154083	25.272	270	4.6
100460:99999	2015	5	1	18	-3.9	2.8827	1020.4	5	2	2.882678977927942	11.591	270	4.6
100460:99999	2015	5	1	17	-3.9	0.31733	1020.4	5	2	0.31733298540789545	16.698	270	4.6
100460:99999	2015	5	1	16	-3.9	1.0751	1020.4	5	2	1.0751264837791352	4.0774	270	4.6
100460:99999	2015	5	1	15	-3.9	2.0494	1020.4	5	2	2.049416247216891	13.215	270	4.6
100460:99999	2015	5	1	14	-3.9	4.9522	1020.4	5	2	4.952187667106246	29.451	270	4.6
100460:99999	2015	5	1	13	-3.9	1.4086	1020.4	5	2	1.408577730613022	27.807	270	4.6
100460:99999	2015	5	1	12	-3.9	4.3249	1020.4	5	2	4.324928015173314	2.1118	270	4.6
100460:99999	2015	5	1	11	-3.9	1.1496	1020.4	5	2	1.1495710221794264	25.509	270	4.6
100460:99999	2015	5	1	10	-3.9	1.5614	1020.4	5	2	1.5613938774247804	2.6761	270	4.6
100460:99999	2015	5	1	9	-3.9	0.15125	1020.4	5	2	0.1512515995438818	4.2795	270	4.6
100460:99999	2015	5	1	8	-3.9	2.6971	1020.4	5	2	2.697138516339501	7.1752	270	4.6
100460:99999	2015	5	1	7	-3.9	1.8016	1020.4	5	2	1.8016284402871308	3.8539	270	4.6
100460:99999	2015	5	1	6	-3.9	4.5698	1020.4	5	2	4.569829092886385	1.281	270	4.6
100460:99999	2015	5	1	5	-3.9	0.0026835	1020.4	5	2	0.0026835082617593375	4.6797	270	4.6
100460:99999	2015	5	1	4	-3.9	4.193	1020.4	5	2	4.192988383638614	18.329	270	4.6
100460:99999	2015	5	1	3	-3.9	4.8035	1020.4	5	2	4.803472407849016	3.0709	270	4.6
100460:99999	2015	5	1	2	-3.9	1.7255	1020.4	5	2	1.7255341579183014	14.111	270	4.6
100460:99999	2015	5	1	1	-3.9	0.045998	1020.4	5	2	0.04599752666885393	20.596	270	4.6
100460:99999	2015	5	1	0	-3.9	0.25412	1020.4	5	2	0.2541232594101578	18.441	270	4.6
100460:99999	2015	4	30	23	-3.9	0.0030714	1020.4	5	2	0.003071375790330566	15.391	270	4.6
100460:99999	2015	4	30	22	-3.9	4.1844	1020.4	5	2	4.184414402672635	3.3233	270	4.6
100460:99999	2015	4	30	21	-3.9	0.80086	1020.4	5	2	0.800864446210463	10.564	270	4.6
100460:99999	2015	4	30	20	-3.9	3.3352	1020.4	5	2	3.3352127982741306	10.193	270	4.6
100460:99999	2015	4	30	19	-3.9	0.45106	1020.4	5	2	0.45105701764617645	10.194	270	4.6
100460:99999	2015	4	30	18	-3.9	1.8461	1020.4	5	2	1.8460615407145036	24.186	270	4.6
100460:99999	2015	4	30	17	-3.9	3.5685	1020.4	5	2	3.5684943135096066	16.496	270	4.6
100460:99999	2015	4	30	16	-3.9	4.7442	1020.4	5	2	4.744197144695919	11.626	270	4.6
100460:99999	2015	4	30	15	-3.9	3.32	1020.4	5	2	3.3199662303014774	22.691	270	4.6
100460:99999	2015	4	30	14	-3.9	0.47089	1020.4	5	2	0.47088679195537775	28.035	270	4.6
100460:99999	2015	4	30	13	-3.9	3.8382	1020.4	5	2	3.838214014919179	4.8504	270	4.6
100460:99999	2015	4	30	12	-3.9	4.84	1020.4	5	2	4.840017521135844	7.8491	270	4.6

Scalability & Performance

- Scalability
  - No single point of failure
  - No special nodes that become the bottle neck
  - Work/data can be re-distributed
- Operational Performance i.e single digit ms
  - Single node for query
  - Single disk seek per query

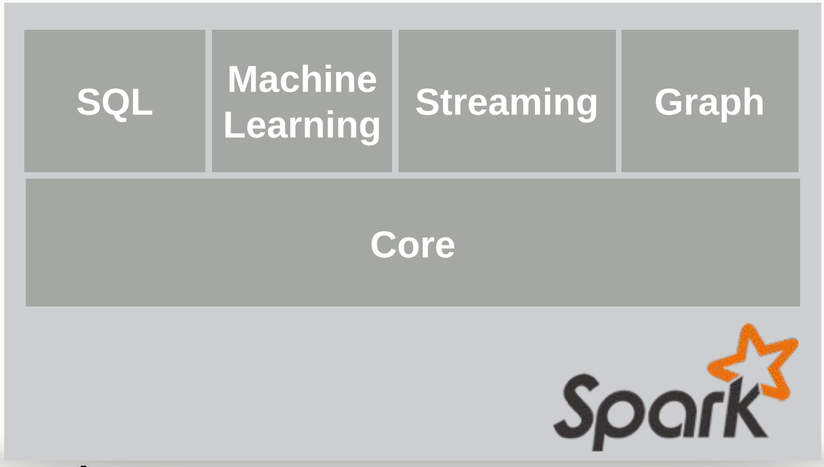

Fault-tolerance
by @jrecursive
So, how do I query the database?
It's not a database. It's a key-value store!
Ok, it's not a database. How do I query it?
You write a distributed map reduce function in Erlang!
Did you just tell me to go myself?
I believe I did, Bob.

Apache Spark

- 10x faster on disk,100x faster in memory than Hadoop MR
- Works out of the box on EMR
- Fault tolerant distributed datasets
- Batch, iterative and streaming analysis
- In memory storage and disk
- Integrates with most file and storage options

Part of most Big Data Platforms

- All Major Hadoop Distributions Include Spark
- Spark Is Also Integrated With Non-Hadoop Big Data Platforms like **DSE**
- Spark Applications Can Be Written Once and Deployed Anywhere

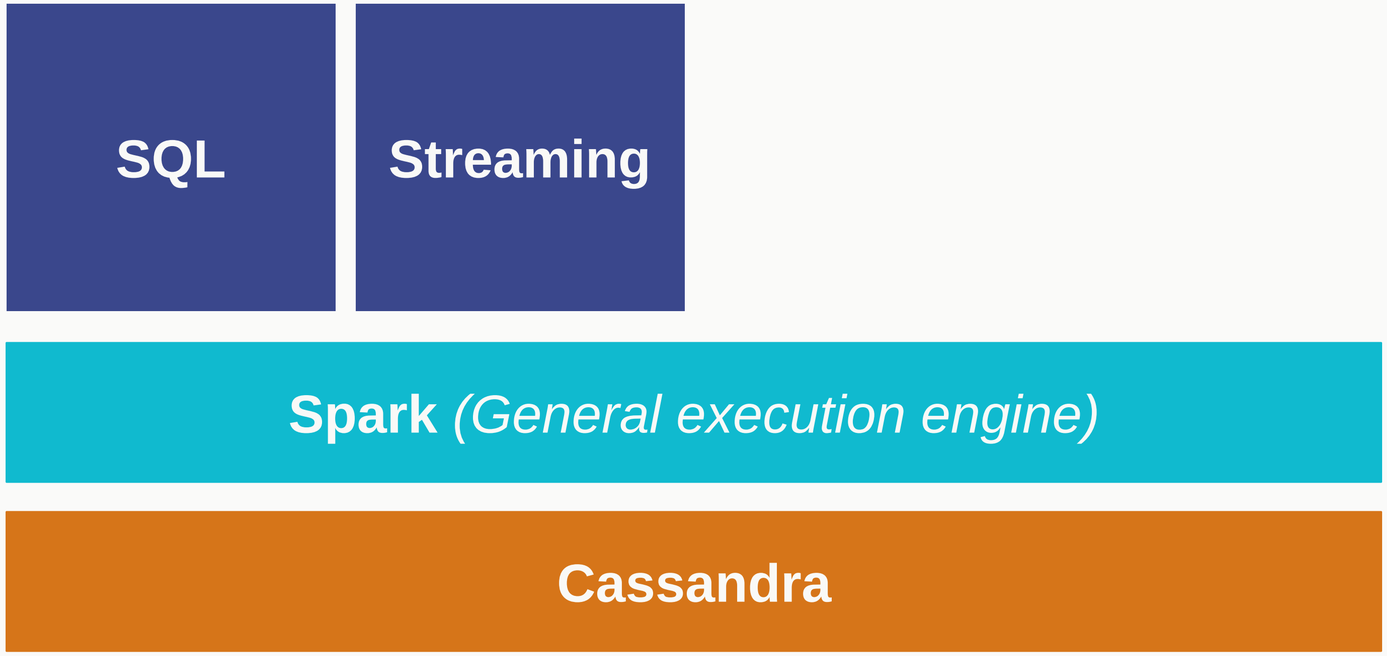


Deploy Spark Apps Anywhere

cloudera MAPR DATASTAX amazon web services™

Hortonworks IBM Pivotal SAP

Components



org.apache.spark.rdd.RDD

- Resilient Distributed Dataset (RDD)
  - Created through transformations on data (map,filter..) or other RDDs
  - Immutable
  - Partitioned
  - Reusable

RDD Operations

- Transformations - Similar to Scala collections API
  - Produce new RDDs
  - filter, flatmap, map, distinct, groupBy, union, zip, reduceByKey, subtract
- Actions
  - Require materialization of the records to generate a value
  - collect: Array[T], count, fold, reduce..

Word count

```
val file: RDD[String] = sc.textFile("hdfs://...")

val counts: RDD[(String, Int)] = file.flatMap(line =>
  line.split(" "))
      .map(word => (word, 1))
      .reduceByKey(_ + _)

counts.saveAsTextFile("hdfs://...")
```

Spark Versus Spark Streaming

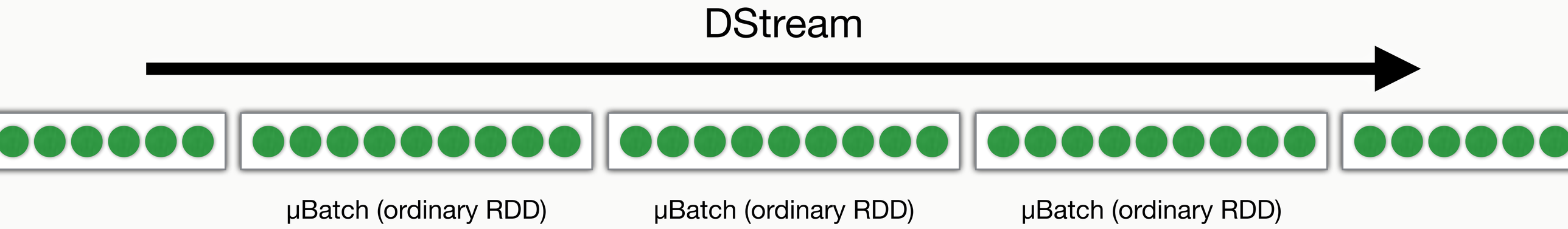
zillions of bytes

gigabytes per second

DStream - Micro Batches

DATASTAX

- Continuous sequence of micro batches
  - More complex processing models are possible with less effort
  - Streaming computations as a series of deterministic batch computations on small time intervals



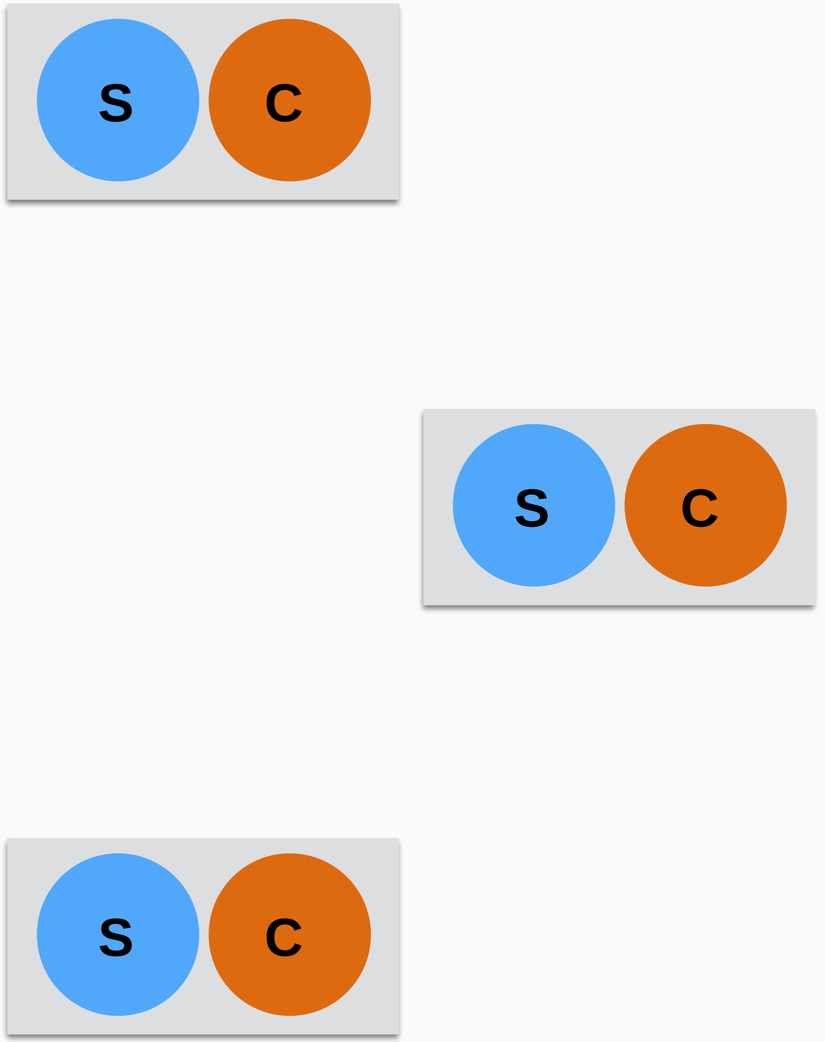
Processing of DStream = Processing of μBatches, RDDs

Spark

Time series example time

Deployment

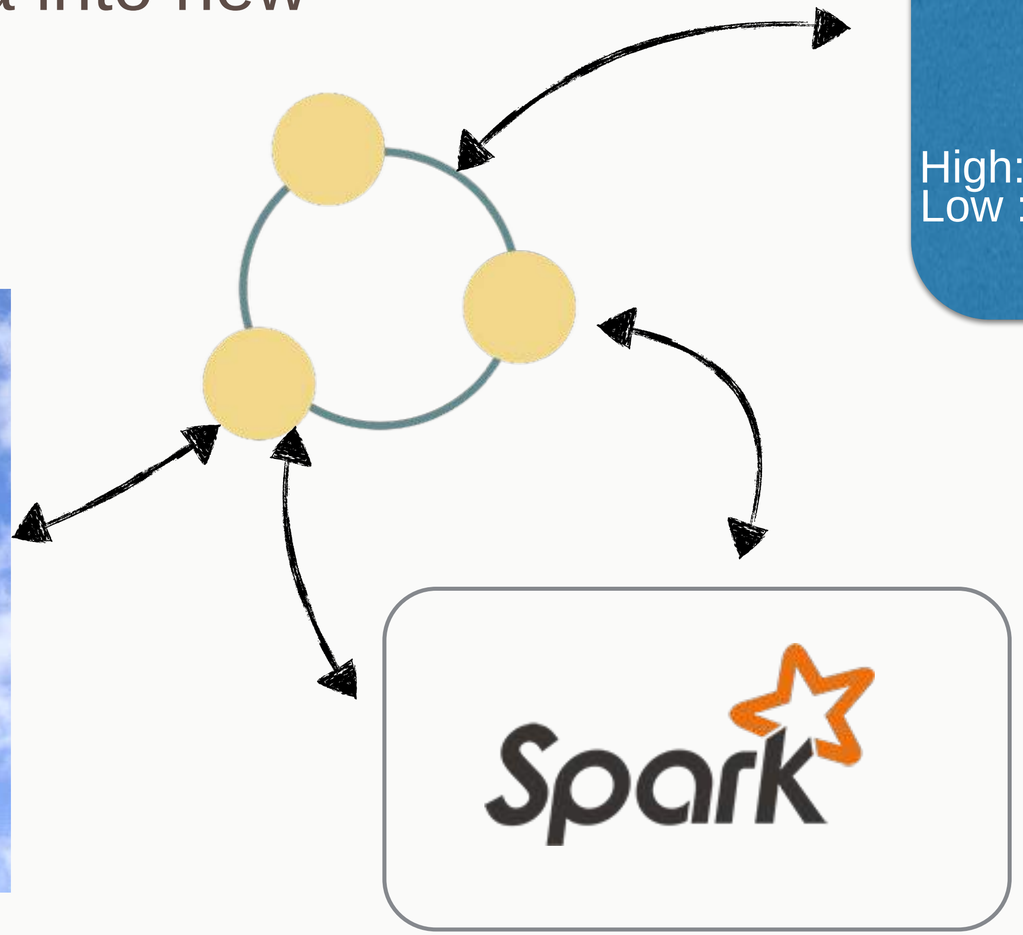
- Spark worker in each of the Cassandra nodes
- Partitions made up of LOCAL cassandra data



Weather Station Analysis

DATASTAX

- Weather station collects data
- Cassandra stores in sequence
- Spark rolls up data into new tables



Windsor California

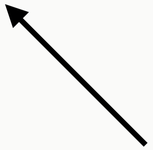
July 1, 2014

High: 73.4F
Low : 51.4F

raw_weather_data

DATASTAX

```
CREATE TABLE raw_weather_data (
    weather_station text,     // Composite of Air Force Datsav3 station number and NCDC WBAN numbe
    year int,                 // Year collected
    month int,                // Month collected
    day int,                  // Day collected
    hour int,                 // Hour collected
    temperature double,        // Air temperature (degrees Celsius)
    dewpoint double,           // Dew point temperature (degrees Celsius)
    pressure double,           // Sea level pressure (hectopascals)
    wind_direction int,       // Wind direction in degrees. 0-359
    wind_speed double,         // Wind speed (meters per second)
    sky_condition int,        // Total cloud cover (coded, see format documentation)
    sky_condition_text text, // Non-coded sky conditions
    one_hour_precip double,    // One-hour accumulated liquid precipitation (millimeters)
    six_hour_precip double,    // Six-hour accumulated liquid precipitation (millimeters)
    PRIMARY KEY ((weather_station), year, month, day, hour)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC, hour DESC);
```



Reverses data in the storage engine.

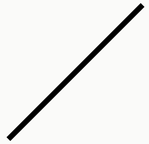
Primary key relationship

DATASTAX

```
PRIMARY KEY ((weatherstation_id),year,month,day,hour)
```

Primary key relationship

```
PRIMARY KEY ((weatherstation_id),year,month,day,hour)
```



Partition Key

Primary key relationship

DATASTAX

```
PRIMARY KEY ((weatherstation_id),year,month,day,hour)
```

Partition Key

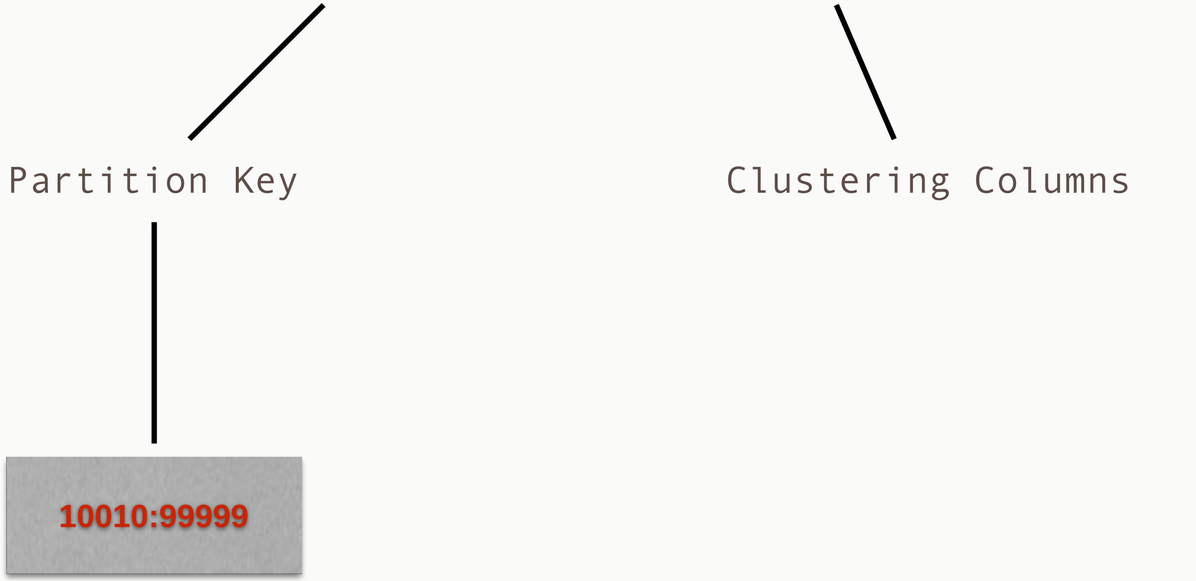
Clustering Columns

```
WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC, hour DESC);
```

Primary key relationship

DATASTAX

```
PRIMARY KEY ((weatherstation_id),year,month,day,hour)
```

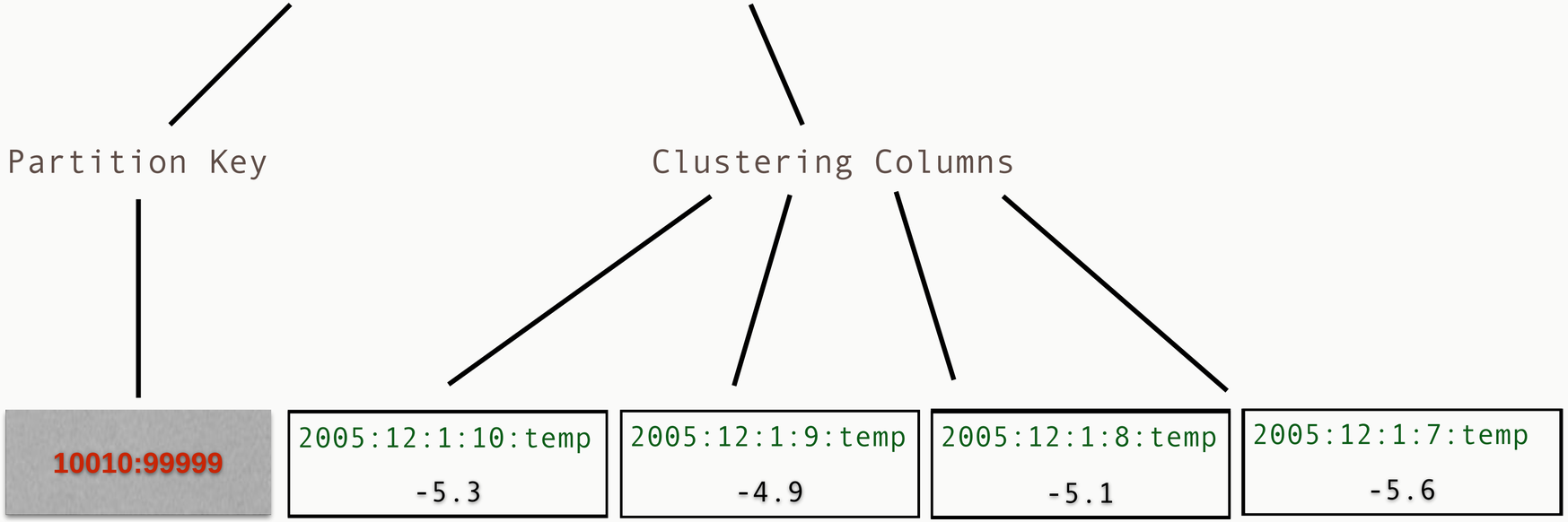



```
WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC, hour DESC);
```

Primary key relationship

DATASTAX

```
PRIMARY KEY ((weatherstation_id),year,month,day,hour)
```

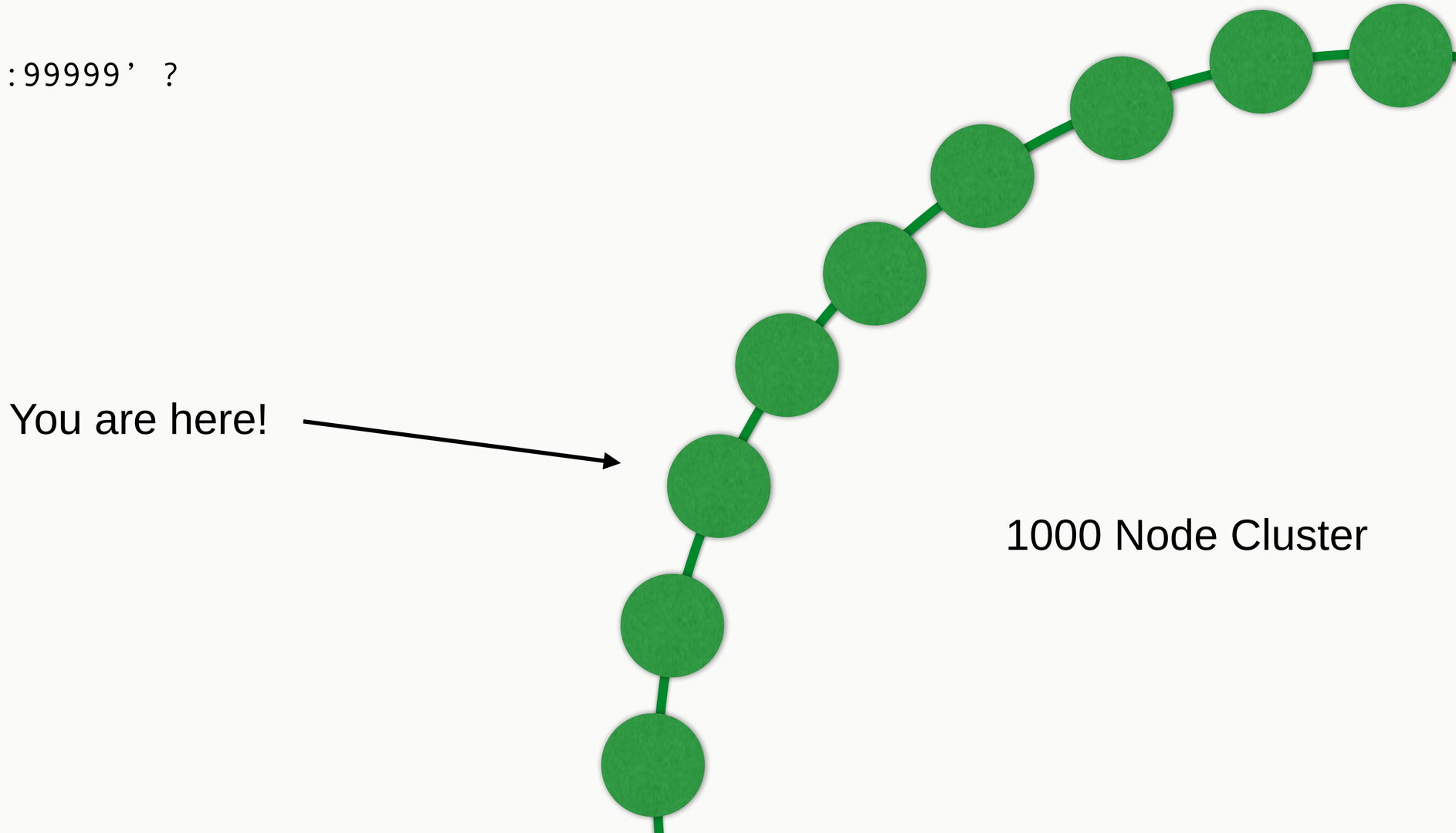



```
WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC, hour DESC);
```

Data Locality

DATASTAX

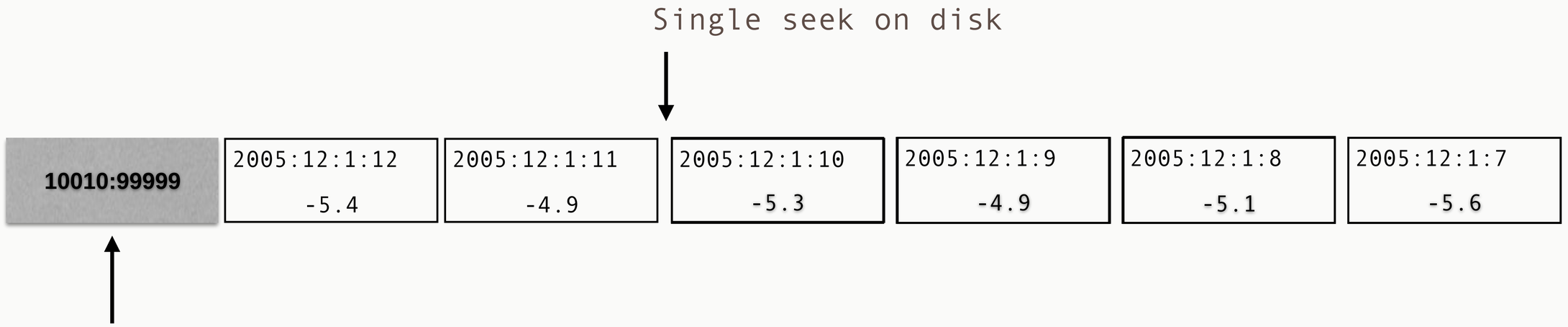
```
weatherstation_id='10010:99999' ?
```



Query patterns

```
SELECT weatherstation,hour,temperature
FROM raw_weather_data
WHERE weatherstation_id='10010:99999'
AND year = 2005 AND month = 12 AND day = 1
AND hour >= 7 AND hour <= 10;
```

- Range queries
- “Slice” operation on disk



Query patterns

```
SELECT weatherstation,hour,temperature
FROM raw_weather_data
WHERE weatherstation_id='10010:99999'
AND year = 2005 AND month = 12 AND day = 1
AND hour >= 7 AND hour <= 10;
```

- Range queries
- “Slice” operation on disk

weather_station	hour	temperature
10010:99999	2005:12:1 10	-5.3
10010:99999	2005:12:1 9	-4.9
10010:99999	2005:12:1 8	-5.1
10010:99999	2005:12:1 7	-5.6

Sorted by event_time

Programmers like this

weather_station

DATASTAX

```
CREATE TABLE weather_station (
    id text PRIMARY KEY,  // Composite of Air Force Datsav3 station number and NCDC WBAN number
    name text,            // Name of reporting station
    country_code text,    // 2 letter ISO Country ID
    state_code text,      // 2 letter state code for US stations
    call_sign text,       // International station call sign
    lat double,            // Latitude in decimal degrees
    long double,           // Longitude in decimal degrees
    elevation double       // Elevation in meters
);
```

Lookup table

daily_aggregate_temperature

DATASTAX

```
CREATE TABLE daily_aggregate_temperature (
    weather_station text,
    year int,
    month int,
    day int,
    high double,
    low double,
    mean double,
    variance double,
    stdev double,
    PRIMARY KEY ((weather_station), year, month, day)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC);
```

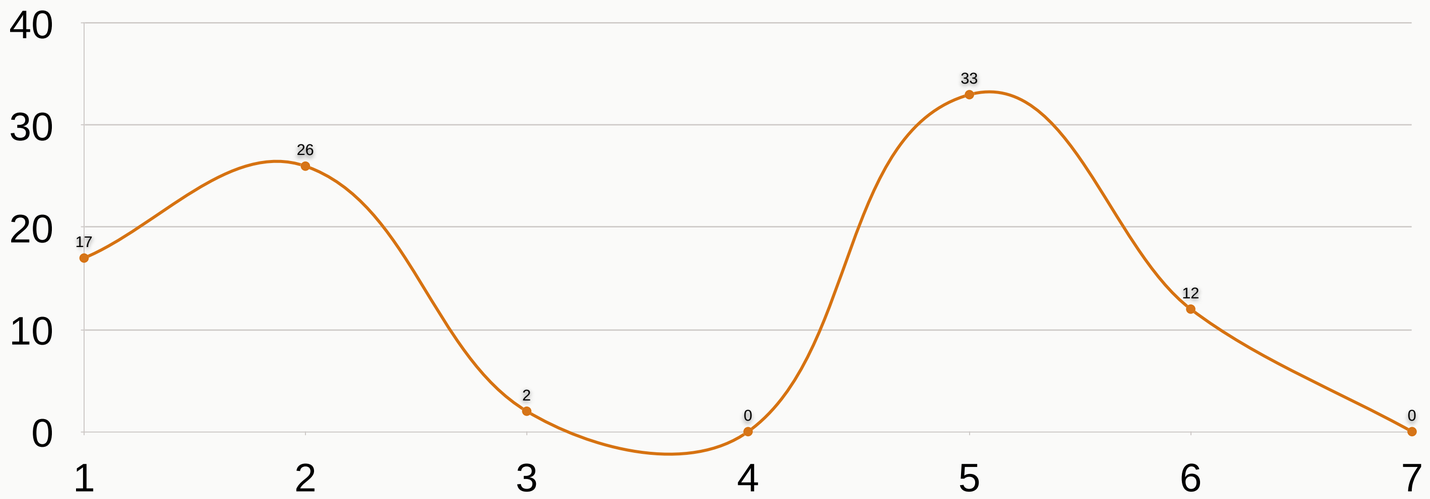
```
SELECT high, low FROM daily_aggregate_temperature
WHERE weather_station='010010:99999'
AND year=2005 AND month=12 AND day=3;
```

```
 high | low
------+------
  1.8 | -1.5
```

daily_aggregate_precip

```
CREATE TABLE daily_aggregate_precip (
    weather_station text,
    year int,
    month int,
    day int,
    precipitation counter,
    PRIMARY KEY ((weather_station), year, month, day)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC);
```

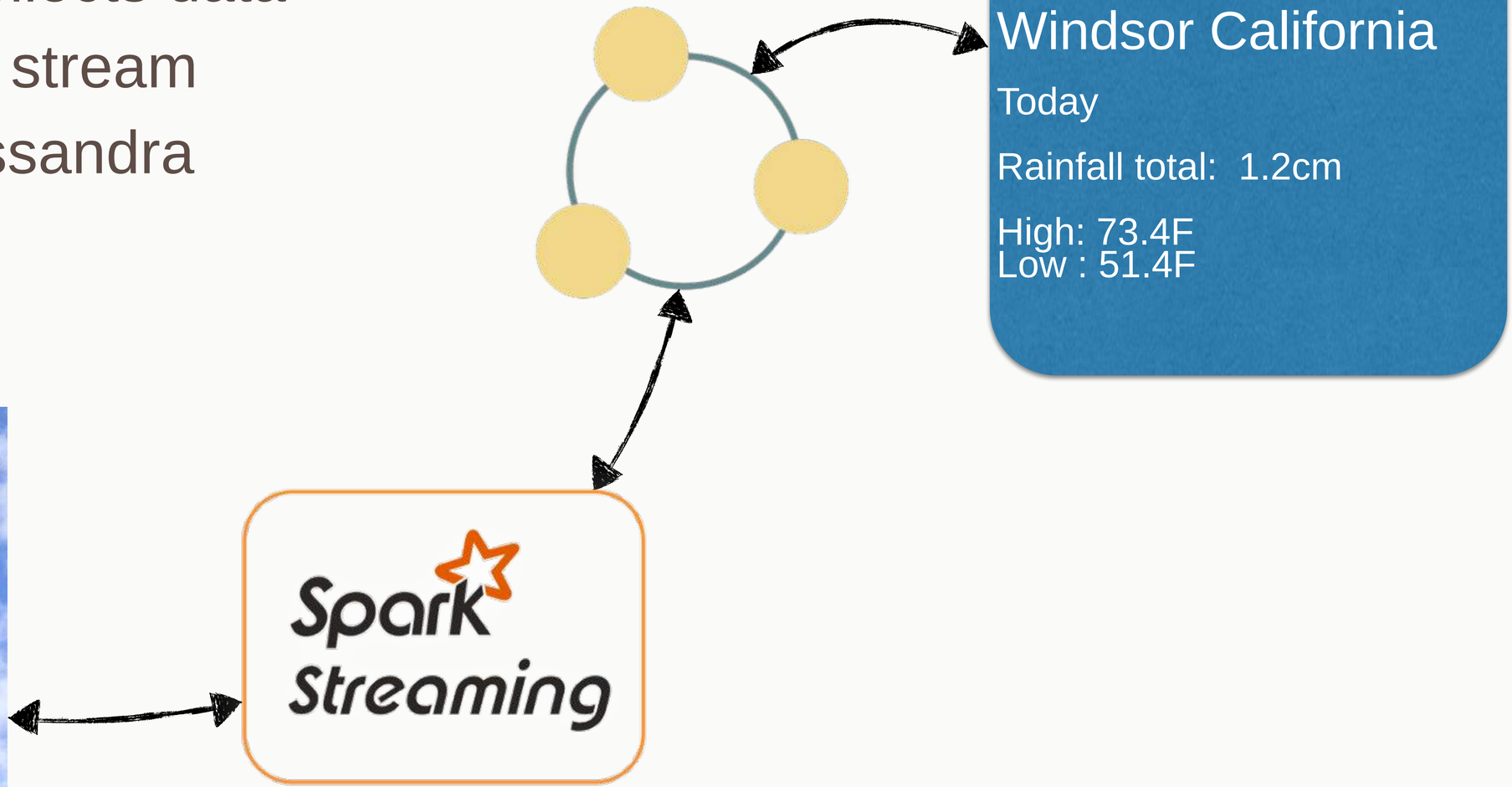
```
SELECT precipitation FROM daily_aggregate_precip
WHERE weather_station='010010:99999'
AND year=2005 AND month=12 AND day>=1 AND day <= 7;
```



Weather Station Stream Analysis

DATASTAX

- Weather station collects data
- Data processed in stream
- Data stored in Cassandra



Incoming data from Kafka

```
725030:14732,2008,01,01,00,5.0,-3.9,1020.4,270,4.6,2,0.0,0.0
```

```
case class RawWeatherData(
  wsid: String,
  year: Int,
  month: Int,
  day: Int,
  hour: Int,
  temperature: Double,
  dewpoint: Double,
  pressure: Double,
  windDirection: Int,
  windSpeed: Double,
  skyCondition: Int,
  skyConditionText: String,
  oneHourPrecip: Double,
  sixHourPrecip: Double) extends WeatherModel
```

Creating a Stream

```
/** Creates the Spark Streaming context. */
lazy val ssc = new StreamingContext(sc, Milliseconds(500))
```

```
val kafkaStream = KafkaUtils.createStream[String, String, StringDecoder, StringDecoder](
  ssc, kafkaParams, Map(KafkaTopicRaw -> 1), StorageLevel.DISK_ONLY_2)
  .map(_._2.split(","))
  .map(RawWeatherData(_))
```

Saving the raw data

```
/** Saves the raw data to Cassandra - raw table. */
kafkaStream.saveToCassandra(CassandraKeyspace, CassandraTableRaw)
```

Building an aggregate

```
CREATE TABLE daily_aggregate_precip (
    weather_station text,
    year int,
    month int,
    day int,
    precipitation counter,
    PRIMARY KEY ((weather_station), year, month, day)
) WITH CLUSTERING ORDER BY (year DESC, month DESC, day DESC);
```

CQL Counter

```
kafkaStream.map { weather =>
  (weather.wsid, weather.year, weather.month, weather.day, weather.oneHourPrecip)
}.saveToCassandra(CassandraKeyspace, CassandraTableDailyPrecip)
```

Batch job on the fly?

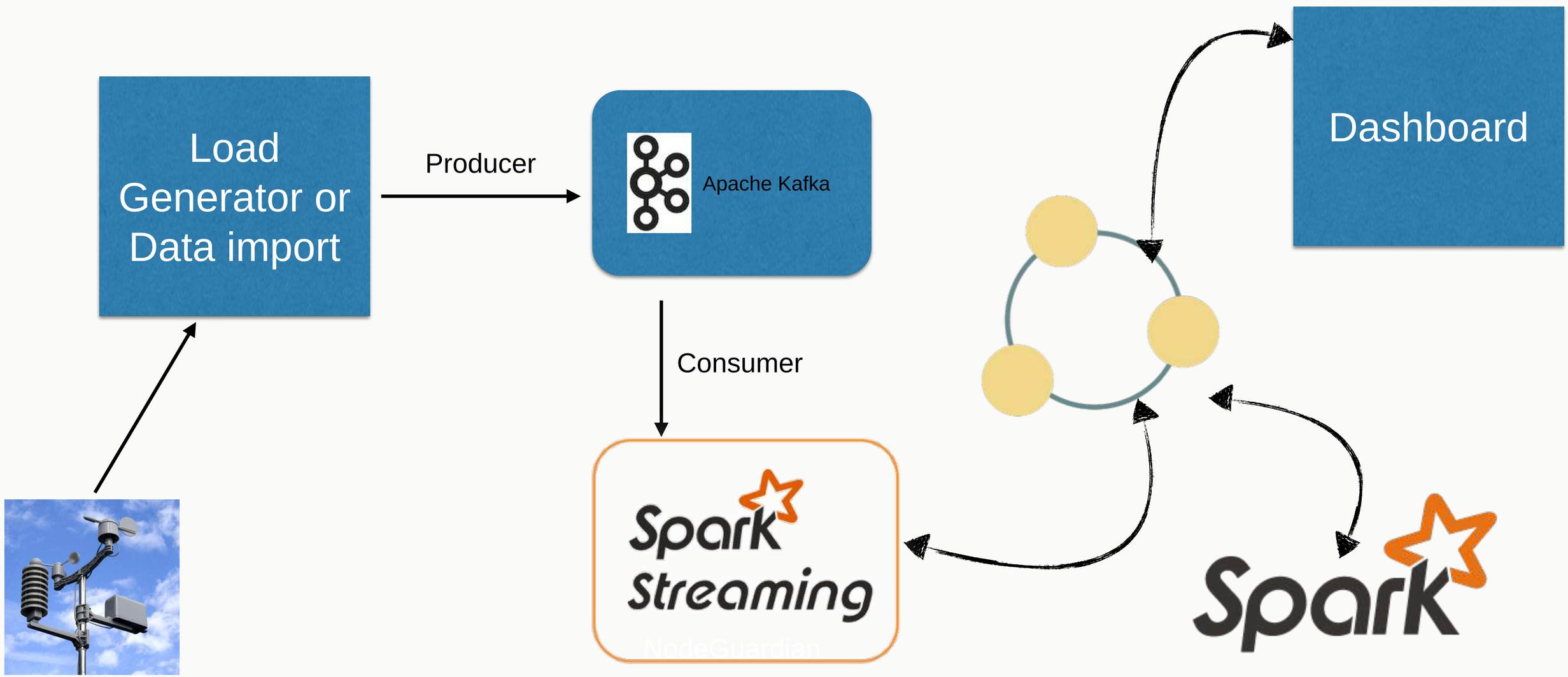
```
val stats: Array[Double] = sc.cassandraTable[Double]("isd_weather_data", "raw_weather_data")
            .select("temperature")
            .where("wsid = ? AND year = ? AND month = ? AND day = ?",
                   "100460:99999", 2015, 4, 30)
            .collect()
val counter = StatCounter(stats)
```

(count: 24, mean: 14.428150, stdev: 7.092196, max: 28.034969, min: 0.675863)

```
val stats: Array[Double] = sc.cassandraTable[Double]("isd_weather_data", "raw_weather_data")
            .select("temperature")
            .collect()
val counter = StatCounter(stats)
```

(count: 11242, mean: 8.921956, stdev: 7.428311, max: 29.997986, min: -2.200000)

Weather data streaming



Summary

- Cassandra
  - always-on operational database
- Spark
  - Batch analytics
  - Stream processing and saving back to Cassandra

Thanks for listening

- Follow me on twitter @chbatey
- Cassandra + Fault tolerance posts a plenty:
  - http://christopher-batey.blogspot.co.uk/
- Cassandra resources: http://planetcassandra.org/